Reduced, Reusable & Reliable Monitor Software

Nicolas F. Rouquette and Daniel L. Dvorak

Nicolas.Rouquette@jpl.nasa.gov, Daniel.Dvorak@jpl.nasa,gov
Jet Propulsion Laboratory
4800 Oak Grove Dr.
Pasadena, CA 90404
http: //www-aig.jpl.nasa.gov

Abstract

This paper presents a software engineering perspective to designing and building fault protection monitor software for spacecraft. We capitalize on fault protection ideas inheritted from Cassini [1] and emphasize streamlining the design and development process on the basis of separating domain-specific monitor specifications from architectural software issues. We emphasize the view of fault protection monitoring as a functional transformation of raw sensor data for feature extraction and symptom detection. Combined with automatic code generation from specification, the functional viewpoint of monitoring can be seen as one application of the cleanroom software engineering methodology [2].

1 Introduction

Sensor monitoring is an integral part of the fault-protection architecture of a spacecraft. Monitors extract features from raw sensor data to detect symptoms of nominal and abnormal behavior. Even though symptom detection is hardware specific, the overall fault protection architecture accommodates a view of monitoring as a data-flow process operating functional transformations of monitor state and sensor data.

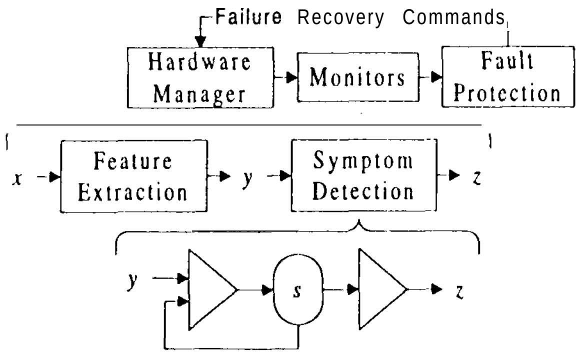

Figure 1: Fault protection architecture

Given sensor data from hardware managers, x, monitors extract salient features, y, used for detecting and reporting symptoms, z, to the fault protection system for diagnosis and recovery [1,4]. Local symptom detection involves updating local state information, s, from feature data, y, and prior state information.

Several generations of spacecraft had severely limited computing resources and consequently could use nothing but basic monitoring techniques such as limit sensing. Faster processors and larger memory can accommodate recent advances in signal processing and fault protection technology. However, sensor monitoring is more than just feature extraction and symptom detection: a spacecraft imposes software architecture standards that all flight software components must follow. This aspect creates several dilemmas: While standardizing software architecture is important, it should not entail freezing monitor design for such designs often need revisions during integration and test. Similarly, accommodating changes due external factors such as hardware revisions should not result re-verifying software architecture compliance.

We approach the problem of designing software monitors from a methodological perspective where sensor monitoring algorithms can be defined, evaluated and selected independently of the specific software architecture standards imposed by the spacecraft design. Our goal is to make basic monitoring capability inexpensive so that the scope of fault protection monitoring is entirely driven from a system engineering analysis instead of being overly constrained from software development concerns. To do so, we first specify each monitor as a dataflow schema of feature extraction and symptom detection operators for reliably detecting and discriminating between nominal and abnormal behavior. Second, we describe all aspects of the software architecture (Figure 2) for sensor monitoring in domain-specific software templates used as patterns for a code generator.

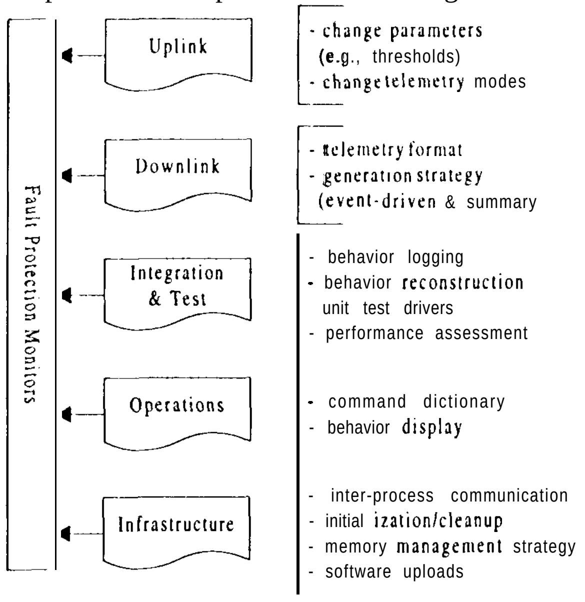

Figure 2: Software Architecture Elements

Finally, all symptom detection algorithms are specified as restricted Harel state-transition diag-rams reusable throughout the spacecraft. The goals of this methodology are to reduce the occurrence of errors through automation, reuse symptom-detection algorithms, and streamline fault protection monitor design and test. This methodology is applied to design and build the fault protection monitors of the Deep Space 1 (1> S-1) spacecraft.

2 Specification-driven monitors

Designing and building fault-protection monitor software for a spacecraft is challenging. Reliability and robustness concerns suggest on one hand freezing specifications, developing the monitors once and testing them thoroughly. On the other hand, the knowledge about possible failures evolves throughout the project thereby changing fault-protection requirements and consequently the fault-protection monitors. Clearly, fault-protection monitors must be designed, built and tested incrementally. An issue is how to accommodate the changes mandated by this incremental approach so that monotonic progress towards the ultimate objectives of reliability and robustness is achieved. Our approach is inspired from the Cleanroom Process Model of software engineering [2] based on evolutionary prototyping and streamlined development from specification to testing [3]. First, we limited the scope of a specification to describe what dataflow processing work must be done in a monitor (Sec. 3). This dataflow specification language works in concert with software architecture templates to describe how a monitor interfaces with the rest of the spacecraft software to achieve its purpose (Sec. 4). We further exploited the separation between software architecture and domain-specific functionality to focus on testing the latter independently from the former and to prevent domain-independent architecture issues from affecting domain-dependent functionality (Sec. 5).

3 Feature extraction & symptom detection

From a high-level perspective, a monitor specification describes the class of nominal and anomalous symptoms relevant for fault protection. From a low-level perspective, a monitor specification describes the data-flow transformations for extracting specific features from raw sensor data and detecting symptoms from such features. It is widely accepted that a careful choice of domain-specific features can greatly simplify the problem of symptom detection [4].

Feature extraction is a domain-specific data transformation function with little potential for reuse. Symptom detection involves a decision-making criterion that may be subject to change, either because of important hardware changes or baseline changes in fault-protection strategy. This function can be already performed in a hardware device or its associated manager in which case there is no additional symptom detection logic. When symptom detection logic is necessary, we define it as a Harel state chart with the restriction that side effects are restricted to modifying the state information associated to that symptom detection algorithm. This restriction allows us to mathematically analyze a state chart diagram as a deterministic function of the data inputs and of the associated state information.

For example, Figure 3 shows the attitude control error monitor used on DS - 1. This monitor tracks the controller performance in the phase plane of the error and rate of error. Traditional control error monitors prescribe a maximum tolerated error (i.e., $e_{min} < e < e_{max}$). This design, inherited from Cassini, is intended to tolerate large errors as long as the controller is working to reduce them (i.e., $e + \wedge \Delta e / \Delta t -$ or $e - \wedge \Delta e / \Delta t +$). The slope of the decision line is proportional to the tolerated time the controller has to reduce an abnormally large error to within the nominal range. This symptom detection logic is inherited from the Cassini spacecraft with the simplification for DS- 1 that the symptom detection threshold logic triggers only on the *f* component of the phase plane as opposed to both for Cassini.

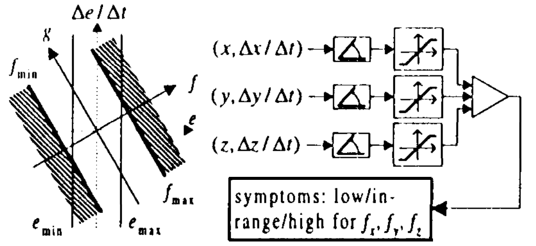

Figure 3: Control error monitor for DS-1.

The specification shown in Figure 3 states that, for each of the three spacecraft axes, the phase plane of the error and rate of error is rotated and that a minimum/maximum threshold symptom detector is applied to each axis of the rotated phase plane. The outputs of the threshold detectors are combined to form the symptom detection message reported to the fault protection engine. This specification is written in a custom dataflow language as described below. Parameters are updated from ground & flight software modules.

```
begin parameters
# threshold = deadband + delta
double delta_x = 0.005
double delta_y = 0.010
double delta_z = 0,005
# Tolerated time outside deadbands implies slope of g axis
double tolerated_time = 3.0
# Delay time before enforcing a tighter threshold.
int transition_time = 60
# Standard monitor tuning parameters
int confidence = 5
int persistence = 3
int decay = 5
end parameters
```

Elements are parameters not visible to ground. They are locally updated by assignment.

```
begin elements
# These time-delay counters hold the number of
# periodic tick remaining before a tighter
# deadband will be honored.
int delay_x = O
int delay_y = 0
int delay_z = O
# Deadbands from the attitude control module (ACM)
double deadband_x = O
double deadband_y = O
double deadband_z = O
end elements
```

Components are instances of Hare] state transition charts. Each chart supports a uniform set of operations for updates, reset, parameter change, and event-driven telemetry generation & reporting as well as telemetry summarization. Each threshold component is specified in terms a min/max range and generic confidence, persistence and decay parameters for the symptom detection logic. Confidence and persistence define how accumulation of evidence is necessary for nominal and abnormal symptom detection respectively. Decay defines a mechanism for aging and eventually ignoring historical data.

```
begin components
# I st component of phase plane: error(x), rate_error(x)
threshold f_x( -(deadband_x+delta_ x), deadband_x+delta_x,
            confidence, persistence, decay)
# 1st component of phase plane: error(y), rate_error(y)
threshold f_y(-(deadband_ y+delta_y), deadband_y+delta_y,
            confidence, persistence, decay)
# I st component of phase plane: error(z), rate_error(z)
threshold f_z(-(deadband_z+ delta_ z), deadband_z+delta_z,
            confidence, persistence, decay)
end components
```

Updates constitute the functional description of feature extraction and symptom detection. Note that messages are reported only if any of the components updated so far necessitate doing so. Also shown is the mechanism for delaying the tightening of ACS deadbands in order to reduce false alarms during the transition.

```
begin updates
function mon_acs_update ( double error[3], double error_rate[3] )
  updates e_edot_2_f(error[ Ql, error_rate[0]) -> f_x
  updates e_edot_2_f(error[ 1], error_ rate[ I ]) -> f_y
  updates e_edot_2_f(error[ 2], error.. rate[2]) -> f_z
  reports MON_ACS_ATT_ERROR ( mon_acs_att_error_t error )
  as error. x <- f_x; error.y <- f_y; error.z <- f_z

function mon_acs_deadbands ( double deadbands[3] )
  if (deadbands[0] > deadband_x)
    then change_ params deadbands[0] -> f_x
    else updates delay_x = transition..time
    and updates deadband_x = deadbands[0]
  if (deadbands[ 1 ] > deadband_y)
    then change_ params deadbands[y] -> f_y
    else updates delay_y = transition..time
    and updates deadband_y = deadbands[ 1]
  if (deadbands[2] > deadband_z)
    then change_params deadbands[z] -> f_z
    else updates delay_z = transition_ time
    and updates deadband_ z = deadbands[2]
  reports MO N_ ACS_ATT_ERROR if changed && needed

for_each tick_period do
  change_ params -> f_x if (delay_x > O) && (--delay_x == O)
  change_ params -> f_y if (delay_y > O) && (--delay_y == O)
  change_params -> f_z if (delay_z > O) && (--delay_z == O)
  report MON_ACS_ATT_ERROR if changed && needed

end updates
# - 'm' is the slope of tbe g-axis (perpendicular to the f-axis).
# - (error,error_rate) is point P in the phase-ptanc coordinates.
# - Line PG is parallel to g-axis and passes through P.
# - (ex,ey) is intersection of line PG with f-axis.
# - Magntiude of f of intersection is sqrt(ex^2 + ey^2)
# - (0,yi) is intersection of line PG with y-axis;
# . sign of yi gives sign of fi.
double e_edot_2_f(double error, double error_ rate)
(
  double m = - l .0 / mon_acs_instance.parameters.tolerated_time;
  double ex = ((m * error) - error_ rate) / (m + (t .0/m));
  double ey = - ex / m;
  double f = sqrt( ex * ex + ey * ey);
  double yi = error_rate - m * error;
  return (yi >= O) '? f: -f;
}
```

4 Software architecture concerns

A monitor specification as earlier described does not describe neither the mechanistic means for feeding raw sensor data to the feature extraction algorithms nor the reporting mechanism for sending symptom reports to the spacecraft fault protection engine. These aspects of monitoring are part of the spacecraft software architecture and design philosophy. Following the cleanroom approach, we isolate all symptom detection algorithms into design-once; reuse-everywhere Harel State charts and group all other software aspects of monitors into architecture-specific templates. The decoupling among architecture concerns, symptom detection and domain knowledge provides a sound basis for reuse and associated code generation techniques.

4.1 *Code generation for state charts*

We use Integrated Systems' Better State Pro for designing state chart diagrams for symptom detection. Better State's code generator produces the C-based flight software as well as Java versions used here for building web-hosted standalone testbeds of each symptom detection algorithm. The web versions have been particularly useful for a comparative and validation study of the monitoring techniques used on the Deep-Space 1 and Cassini spacecraft [5].

Without a state chart code generator, the quality of the symptom detection algorithms would clearly be subject to the quality of the code hand-produced. An automatic code-generation capability helps eliminates a large class of human-induced errors in flight software. However, the cleanroom approach could be pushed a step further to eliminate errors at the level of state charts thereby strengthening even more the quality of the resulting flight software. Since symptom detection algorithms must follow specific conventions for data processing and symptom reporting, we can already state several properties that all state charts representing such algorithms must meet.

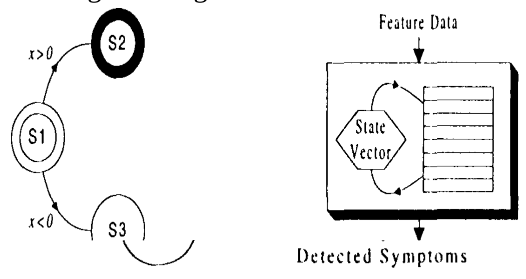

Figure 4: Limitations of State Charts.

For example, Figure 4 (left) shows a state chart with two design errors: 1) if $x=0$, then no transition is made, and 2) there is no terminal state following the transition to S3 so the state machine is stuck in S3. Current code generation techniques use a state vector to represent the progress of the computation through state transitions. This technique is adequate for arbitrary state charts; however, it is too general for state charts that are equivalent to pure functions of state (right). For example, the explicit state-based representation used in the generated code could be eliminated entirely by calculating, at generation time, the functional composition of the actions and tests of the finite set of paths through the state chart. Similarly, the functions and structure of the state chart could be exploited for test generation and analysis purposes much like it is done in VLS1 chip design.

4.2 *Code generation for specifications*

This code generator started as a much less ambitious effort to save manual software development. The importance of the generator rapidly grew to a much more comprehensive role once the cleanroom process benefits of increased productivity and reduced development costs far outweighed the development of the generator itself.

The separation of software architecture design and domain-specific monitor specification allows us to decouple the schedules and workforce necessary to handle design and development issues about software architecture on one hand and fault protection on the other; both of which independently affect how monitor specifications are written. In practice, software architecture matters affect fault protection issues and vice versa. Resolving such issues pertains to spacecraft design, which is beyond the scope of this paper. However, telemetry is one area where this interdependence is not resolved by design but still persists until late in integration and test.

A spacecraft requires several levels of telemetry granularity to provide ground operators the flexibility of adjusting the level of spacecraft visibility to their needs. The lowest level of insight comes from raw sensor data fed to the monitors. Since ground operators need the flexibility to change which monitor channels will be scrutinized for anomaly analysis, all monitor input channels are fitted with input ring buffers. The signature of an anomaly then dictates which subset of the ring buffers will be relevant for ground analysis and therefore must be downlinked via recorded telemetry. To accommodate fault protection changes without software updates, we encoded in parametric form all domain-specific decisions about monitor telemetry generation and recording. That is, the software templates for code generation determine where parametric control of telemetry generation and recording is applicable while the monitor specification implicitly defines the domain-specific nature of that telemetry.

In the absence of failures, monitor telemetry can be generated either in a fixed or variable (event-driven) format. For DS- 1, fixed telemetry packets summarize the monitor activity since the last ground communication pass while event-driven telemetry supports ground testing and real-time spacecraft operation. The architectural split between event-driven and packetized telemetry stems from our experience in unit testing and behavior reconstruction as described next.

5 Unit testing & reconstruction

The unit testing of the monitor software has been performed on Unix workstations to leverage the rich development tools available. For example, a logging mechanism is added to the software architecture template files to generate a record of just enough monitor activity to have full visibility into all of the internal decision-making pertaining to symptom detection. Since a real-time testbed is practically inadequate for analyzing the individual behavior of monitors and of other tasks, we have extended the logging capability to allow the full behavior reconstruction of the monitors outside the testbed. This is achieved by recording the input stimuli seen by the monitors in the testbed. Then, a test driver emulates the testbed interfaces the monitors have and replays the recorded stimuli to reproduce the same behavior that occurred in the testbed.

This simple technique has proven to be quite useful for the detailed analysis of integration problems between monitors and other software modules such as hardware simulators, device managers and fault protection software. In practice, behavior reconstruction helps us reduce integration problems to either an incorrect monitor specification or a software issue outside the monitors.[1] Behavior reconstruction is typically seen as a helpful technique for integration. We have extended this technique to evaluate designs and solutions for tackling event-driven telemetry generation as well as interface verification.

Event-driven telemetry provides visibility into the activities of the spacecraft for the purpose of ground-based testing and interaction with the spacecraft. Since there is not enough bandwidth to instrument all sensor inputs, some summarization is necessary to fit within the available bandwidth and to convey the essential information. To avoid introducing errors in computing summaries, we rationalize how this process must be done on the basis of the monitor specification itself. Each symptom detection component (e.g., a threshold) has several internal logical states which remain unreported (e.g., not enough confidence in nominal data or not a persistent abnormality) while some are reported as nominal indicators (e.g., within range) and others as abnormal symptoms (e.g., too high, too low). Thus, for each symptom detection component, we associate a histogram table summarizing the occurrence of states seen and generate telemetry as follows: whenever a component enters a reported state[2], the histogram table is examined and all non-zero entries are reported in the real-time telemetry stream. This approach provides the starting point to perform a temporal segmentation of the raw sensor data in terms of intervals corresponding to nominal, abnormal and unknown behavior. These interval labels could be used to further bias the summarization techniques for detecting underlying drifts and evaluating the need of monitor parameter corrections.

In a spacecraft, the monitors are part of an overall fault protection architecture such as [6]. To eliminate software integration issues with high-level fault protection software, we need to validate and verify the overall fault protection capability as a functional unit, not as a set of modules each requiring its own separate integration and tests. To help address this problem, we use behavior reconstruction to reasonably enumerate the possible interactions originating from either monitors or fault protection software. On the monitor side, we generate behavior scripts to enumerate all possible independent causes of symptom detection. For this enumeration to be exhaustive, we use the monitor specifications to trace and test all possible data pathways from raw sensor data to any detected symptom. On the fault protection side, we look at all of the properties that the fault protection models are predicated on[3] and examine all of the possible ways in which these symptoms can be

[1] This assumes that 1) the monitor software architecture is consistent with that designed for the spacecraft and that 2) the software templates have been validated accordingly prior to integration.

[2] Reported states affect upper layers of fault protection and therefore must be externally visible.

[3] This analysis is *greatly* simplified if high-level fault protection is implemented with a rigorous theoretical basis (e.g. [6]) as opposed to an ad-hoc design that requires extensive analysis for validation.

generated from the monitors. While these two tests are empirical in nature, checking their mutual consistency and exhaustiveness provides an important basis for validating the overall fault protection monitors and models.

6 Discussion

6.1 *Status*

The methodology presented here was created as an on-going process improvement during the course of designing and building the fault protection monitors for the Deep Space One spacecraft scheduled for launch in July 1998 [7]. To date, we have designed, built and unit tested 5 symptom detection algorithms. There are 159 instances of these detectors among the 11 monitor specifications written.

6.2 *Cleanroom methodology*

Initially, we saw monitor specifications and specification-driven code generation as a basis for accommodating the separate and conflicting schedules of software development and of system-level fault protection engineering. However, monitor specifications soon became useful to generate a whole class of related products from test drivers to behavior reconstruction analyzers and from ground monitor commands to downlink monitor telemetry. This approach makes sense because we precisely define the nature of the fault protection monitors as a mathematical function transforming raw sensor data into symptoms of nominal and abnormal behavior. This formal view of software echoes the emphasis of transforming specifications into mathematical functions operating on data as advocated in cleanroom engineering [8,2,3] and formal approaches to software design and validation [9]. In fact, this functional view provides the basis for establishing a simple architecture for monitoring (Figure 1) and, more importantly, the rationale for incorporating in that dataflow architecture all the software requirements that fault protection must meet (e.g., initialization, telemetry, commanding, etc.). Doing so simplifies the process of validating fault protection monitors in terms of domain-specific content and architecture. The savings stem from factoring out the domain-independent architectural aspects of monitoring from the domain-specific details captured in the monitor specifications. For example, instead of following the statistical usage testing approach advocated in [8], we use the functional structure of the monitor specification to first determine what are the independent functions implied by the specification and, second, to test such functions thoroughly and exhaustively if possible.

6.3 *Future work*

The methodology presented here represents work in progress targeted at Deep Space One in particular. Additional extensions are necessary to automate the validation of software architecture templates to make the method more accessible. We have not addressed how to incorporate adaptive monitoring techniques such as [10].

7 Acknowledgements

Douglas Bernard, Pandu Nayak, Brian Williams, Bill Millar, Jim Kurien and Sanford Krasner all provided significant feeback on this work.

The research described in this paper was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

References

1 G. M. Brown, D. Bernard, R. Rasmussen, "Attitude and Articulation Control for the Cassini Spacecraft, A Fault Tolerance Overview", 14* AIAA/IEEE Digital Avionics Conference, 1995.

2 R. Linger. "Cleanroom Process Model", IEEE Software, Vol. 11, pp. 50-58, 1994.

3 M. Deck. "Cleanroom Review Techniques for Application Development", 6* Int.Conf.On Software Quality, Ottawa, Canada, 1996.

4 M. Basseville, "Detecting Changes in Signals and Systems - A survey", Automatic, Vol. 24, No. 3, pp. 309-326, 1988

5 R. Lutz, "Reuse of a Formal Model for Requirements Validation", draft paper, JPL internal document, 1997.

6 B. Williams and P. Nayak, "A Model-based Appreach to Reactive Self-Configuring Systems", Proc. of AAAI-96, 1996.

7 D. Bernard, B. Pen, "Designed for Autonomy: Remote Agent for the New Millennium Program", iSAIRAS-97.

8 H. D. Mills, "Zero-Defect Software: Cleanroom Engineering", Advances in Computers, Vol. 36, pp. 1-41, 1993.

9 M. Stickel *et al.*, "Deductive composition of astronomical software from subroutine libraries", 12th Conf. On Automated Deduction, 1994.

10 D. DeCoste, "Automated Learning and Monitoring of Limit Functions", *i*-SAIRAS, 1997.